Real-Time k-bounded Preemptive Scheduling

Sivan Albagli-Kim* Baruch Schieber† Hadas Shachnai* Tami Tamir‡

Abstract

We consider a variant of the classic real-time scheduling problem, which has natural applications in cloud computing. The input consists of a set of jobs, and an integer parameter $k \geq 1$. Each job is associated with a processing time, a release time, a due-date and a positive weight. The goal is to feasibly schedule a subset of the jobs of maximum total weight on a single machine, such that each of the jobs is preempted at most k times.

Our theoretical results for the *real-time k-bounded preemptive scheduling* problem include hardness proofs, as well as algorithms for subclasses of instances, for which we derive constant-ratio performance guarantees. We bridge the gap between theory and practice through a comprehensive experimental study, in which we also test the performance of several heuristics for general instances on multiple parallel machines. We use in the experiments a linear programming relaxation to upper bound the optimal solution for a given instance. Our results show that while k-bounded preemptive scheduling is hard to solve already on highly restricted instances, simple priority-based heuristics yield almost optimal schedules for realistic inputs and arbitrary values of k.

1 Introduction

1.1 Background and Problem Statement The emergence of cloud systems as a common computation resource gives rise to plenty of optimization problems, many of which deal with scheduling jobs that require cloud services. Such jobs are often business critical and time-sensitive, mandating strict service level agreements on completion time. Thus, cloud systems are required to support *real-time scheduling*, where each job must be processed within a given time-window.

Preemption is a key factor in real-time scheduling, since it allows the operating system to immediately allocate the processor to incoming jobs requiring urgent service. In fully preemptive systems, a running job can be interrupted at any time by another job with higher priority, and be resumed to continue when all higher priority jobs have completed. In other systems, preemption can be completely forbidden to avoid unpredictable interference among jobs. The question of enabling or disabling preemption during task execution has been widely investigated (see, e.g., [21, 6, 9], and the comprehensive survey in [8]).

While non-preemptive scheduling introduces additional blocking time in higher priority jobs, in many practical scenarios (e.g., I/O scheduling or communication in a shared medium), preemption is either impossible or prohibitively expensive [11]. This calls for a *bounded-preemptions* scheduling model, which allows to reduce costs and runtime overhead caused by preemptions.

In this model, we study the following natural problem. Let $\mathcal{J}$ be a set of n jobs, available for processing on a single machine, which can process at most one job at any time. Each job $j \in \mathcal{J}$ is associated with a triple $\langle r_j, d_j, p_j \rangle$, denoting the job release time, deadline, and processing time. Also, each job j has a positive weight w_j. We are also given an integer parameter $k \geq 1$. The objective is to feasibly schedule on the machine a subset of the jobs of maximum total weight, such that each of the jobs is preempted at most k times.

We also consider a variant of the problem with multiple machines. We note that in a feasible schedule on multiple parallel machines, the processing of a job may consist of at most $(k+1)$ disjoint segments, which can be arbitrarily distributed among the machines. For example, in a feasible 2-bounded schedule on $m = 2$ machines, a job may start processing on one machine, then migrate to proceed running on the second machine, and complete its execution on the first machine. Such scenarios beg the following question: "Are migrations essential for maximizing system throughput?" We explore the power of migrations in the bounded-preemptive scheduling model, by comparing heuristics that allow migrations to those in which jobs may be preempted, but must resume their processing on the same machine (see Section 5).

*Computer Science Department, Technion, Haifa 3200003, Israel. E-mail: {hadas,sivanal}@cs.technion.ac.il. This work was partly carried out during a visit to DIMACS supported by the National Science Foundation under grant number CCF-1144502.

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA. E-mail: sbar@us.ibm.com

‡School of Computer Science, The Interdisciplinary Center, Herzliya, Israel. E-mail: tami@idc.ac.il

1.2 Our Results The contribution of this paper is two-fold. Our main theoretical contribution includes hardness results as well as approximation algorithms for the problem. We complement these results by extensive experimental analysis, bridging the gap between theory and practice.

We first study (in Section 3) the computational complexity of our problem. Specifically, we show that real-time k-bounded preemptive scheduling is NP-hard already for uniform-length job instances with unit weights for $k = 1$, and strongly NP-hard for $k > 1$. Thus, we establish for this subclass, of uniform-length unit-weight instances, an interesting distinction between the hardness status of non-preemptive, or (unbounded) preemptive scheduling, which are both solvable in polynomial time, and bounded-preemptive scheduling. For more general instances, our results show that while unbounded preemptive scheduling is solvable in pseudo-polynomial time, real-time scheduling with bounded number of preemptions is as hard as the classic non-preemptive real-time scheduling problem.

In Section 4, we present constant factor approximation algorithms for two subclasses of instances. This includes a $\frac{k+1}{k}$-approximation algorithm for general instances with the same (unit) weights and any $k \geq 1$, and a 4-approximation algorithm, Alg_{MU}, for the maximum utilization version of the problem, in which the weight of each job is equal to its processing time.

We evaluate the performance of our algorithms in practice through a comprehensive experimental study, in which we also test the performance of several heuristics for general instances on multiple parallel machines. Our experimental results (in Section 5) show that while k-bounded preemptive scheduling is hard to solve already on highly restricted instances, simple priority-based heuristics yield almost optimal schedules for realistic inputs and arbitrary values of k.

The two heuristics that outperform the greedy scheme (see Section 5.1) assign jobs on idle-intervals that are less likely to be requested by other jobs. Specifically, a job j not considered yet generates load $l_j = \frac{p_j}{d_j - r_j}$ in the interval $[r_j, d_j)$. Thus, each idle-interval of the machine is associated with a *demand*, given by the sum of loads on this interval. Our heuristics give higher priority to intervals with lower demand, justified by the fact that the demand for other idle-intervals remains high also when excluding the currently considered job. Our experiments show that the performance of these heuristics is close to the optimal. The implementation of the heuristics is simple and scalable; thus, we believe they will prove useful in solving other utilization problems.

We compare the performance of our heuristics to an optimal solution, by using a non-trivial *linear programming (LP)* relaxation, whose solution yields an upper bound for the optimum (see Section 2). Due to the large size of the program, we could solve it for relatively small instances. These instances were carefully generated, to maintain the characteristics of large instances, through adjustment of job processing times and densities (see Section 5). Comparing the performance of Alg_{MU} to the LP solution, we conclude that in practice our algorithm yields performance guarantee much better than the theoretical bound of 4.

Finally, our study offers interesting insights into the power of migration on parallel machines (see Section 5.2). Specifically, we show that as k grows larger, the power of migration (subject to a threshold parameter) becomes predominant, in particular, for 'challenging' job instances (see Figure 6).

1.3 Related Work The non-preemptive real-time scheduling problem, which has been widely studied, is known to be NP-hard in the strong sense, even if all jobs have the same (unit) weights [10]. The best known approximation ratio of 2 is due to Bar-Noy et al. [5, 4]. Several branch and bound algorithms were proposed for the unweighed version of the problem [3, 13, 19].

Some special cases are solvable in polynomial time. In particular, when all jobs have the same (unit) weight and the same release time, non-preemptive real-time scheduling can be solved in $O(n \log n)$ time, using Moore's algorithm [18]. Also, when release times and due dates of jobs are ordered similarly (i.e., $r_i < r_j \Rightarrow d_i \leq d_j$), the problem is solvable in $O(n^2)$-time using dynamic programming [15]. An improved $O(n \log n)$-time algorithm was proposed by Lawler [16]. The problem with uniform-length jobs and unit weights can be solved in $O(n^3 \log n)$ steps [12]. Baptise [1] proposed a strongly polynomial algorithm for the weighted version with uniform-length jobs. Other polynomially-solvable subclasses can be found, e.g., in [7].

The problem of (unbounded) preemptive real-time scheduling was extensively studied as well. It is known to be NP-hard, but can be solved in pseudo-polynomial time, using an algorithm of Lawler [17]. In the unweighed case, the algorithm becomes strongly polynomial. Baptiste gave a strongly polynomial time algorithm for the weighted problem with uniform processing times [1, 2]. These results leave open the complexity of our real-time bounded-preemptive scheduling problem.

There has been earlier work on bounded preemptive scheduling in the parallel machine environment. The paper [20] considered the *generalized multiprocessor scheduling (GMS)* problem, in which there is a total bound on the number of preemptions throughout a fea-

sible schedule, and the goal is to find a schedule that satisfies the preemption constraint, such that the maximum job completion time (or, makespan) is minimized. The paper presents approximation schemes for identical and uniform machines.

Real-time bounded-preemptions scheduling has been studied earlier from a practical viewpoint (see, e.g., [21, 6, 9] and the references therein); however, the underlying assumptions and the objective functions were different than ours. For example, preemptions may be associated with a cost, or allowed only during a specific period throughout the time-window of a job.

To the best of our knowledge, the real-time k-bounded preemptive scheduling problem is studied here for the first time.

2 A Linear Programming Formulation

We describe below a *linear programming (LP)* formulation of the *k-bounded preemptive scheduling* problem, in which the integrality constraints are relaxed. We use this program in our empirical study to obtain an upper bound on the value of an optimal schedule for a given instance. Let x_j be an indicator for the selection of job j; that is, $x_j = 1$ if j is scheduled, and $x_j = 0$ otherwise. Let $L = \max_{1 \le j \le n} d_j$ be the latest due-date of any job. For each job j and $t \in [r_j, d_j + 1)$, let $x_{j,t}$ be an indicator variable for processing job j at time t; that is, $x_{j,t} = 1$ if j is processed during the time slot $[t, t+1)$.

In the integer programming formulation of our problem, Constraints (2.6) and (2.7) are $x_j \in \{0, 1\}$ and $x_{j,t} \in \{0, 1\}$, respectively. Thus, a schedule of a job corresponds to a binary sequence of length L, describing the time slots in which the job is processed. Constraints (2.2) ensure that the machine processes at most one job in any time slot. Constraints (2.4) ensure that each job is processed before its deadline. Constraints (2.5) correspond to the bounded number of preemptions. Specifically, each job j can move from the status of being processed to not-processed at most $k+1$ times. This implies that the number of preemptions j experiences is at most k. Note that $x_{j,t}$ is defined also for $t = d_j$, for all $1 \le j \le n$. By Constraints (2.4), these variables are set to 0. They are only used to count the completion of job j, which may occur at its deadline.

To eliminate the *max* operation, we add a variable $z_{j,t}$ for any $t \in [r_j + 1, d_j]$ and replace (2.5) by three sets of constraints (2.8), (2.9) and (2.10).

3 Computational Complexity

In this section we study the computational complexity of real-time k-bounded preemptive scheduling. Recall that for uniform-length jobs, real-time scheduling is solvable on a single machine in polynomial time when

$$\text{maximize} \quad \sum_{j=1}^{n} w_j x_j$$

subject to:

$$x_{j,t} \le x_j \qquad \forall j, t \tag{2.1}$$

$$\sum_{j=1}^{n} x_{j,t} \le 1 \qquad \forall t \tag{2.2}$$

$$\sum_{t} x_{j,t} \ge p_j \cdot x_j \qquad \forall j \tag{2.3}$$

$$x_{j,t} \le 0 \qquad \forall j, t : t \ge d_j \tag{2.4}$$

$$\sum_{t \in [r_j+1, d_j]} \max\{x_{j,t-1} - x_{j,t}, 0\} \le k+1 \qquad \forall j \tag{2.5}$$

$$0 \le x_j \le 1 \qquad \forall j \tag{2.6}$$

$$0 \le x_{j,t} \le 1 \qquad \forall j, t \tag{2.7}$$

$$\sum_{t \in [r_j+1, d_j]} z_{j,t} \le k+1 \qquad \forall j \tag{2.8}$$

$$z_{j,t} \ge x_{j,t-1} - x_{j,t} \qquad \forall j, t \tag{2.9}$$

$$z_{j,t} \ge 0 \qquad \forall j, t \tag{2.10}$$

preemptions are either not allowed [1] or unlimited [17]. We show that bounded-preemptive scheduling is NP-hard. Our reductions rely on scheduling instances containing *tight* jobs, namely, jobs j for which $p_j = d_j - r_j$. The existence of tight jobs adds constraints on the schedule of the remaining jobs, that should utilize the idle-segments left on the machine after the assignment of tight jobs. We obtain the following results.

THEOREM 3.1. *The problem of real-time k-bounded preemptive scheduling with uniform-length jobs is NP-hard for $k = 1$, and strongly NP-hard for $k > 1$.*

Proof. Clearly, in a feasible k-bounded preemptive schedule, the processing of each job splits into at most $k+1$ parts. Let $k' = k+1$. For $k > 1$, we show a reduction from k'-Partition. Given a set of $k'm$ numbers $\mathcal{A} = \{a_1, ..., a_{k'm}\}$ and a number Z, such that $\forall i, a_i \in (\frac{Z}{k'+1}, \frac{Z}{k'-1})$ and $\sum_{i=1}^{k'm} a_i = Zm$, the goal is to divide $\mathcal{A}$ into m subsets $S_1, \cdots, S_m \subseteq \mathcal{A}$ such that, for all $1 \le j \le m$, $\sum_{a \in S_j} a = Z$. The k'-Partition problem is known to be strongly NP-complete for any $k' > 2$ [10].

Given an instance of k'-Partition, we construct the following input for real-time k-bounded preemptive scheduling. We have $(k'+1)m$ jobs, all have the same

processing time, Z. There are $k'm$ tight jobs, for each of which $d_j - r_j = Z$, and m jobs for which $r_j = 0$ and $d_j = \sum_j p_j = (k'+1)mZ$. Specifically, for $1 \leq j \leq k'm$, the window of job j is $[\sum_{i=1}^{j}(a_i)+(j-1)Z, r_j+Z)$, and $p_j = Z$. For each $k'm+1 \leq j \leq (k'+1)m$: the window of job j is $[0,(k'+1)mZ)$, and $p_j = Z$.

For example, consider an instance for 3-Partition with $\mathcal{A} = \{6,7,8,8,8,9\}$, and $Z = 23$. The corresponding instance refers to $k = 2$ and $m = 2$. There are six tight jobs that must be placed in $[6,29),[36,59),[67,90),[98,121),[129,152)$ and $[161,184)$, and two jobs that need to be scheduled in $[0,184)$, each in at most 3 pieces. Note that the tight jobs leave idle-segments of lengths $6,7,8,8,8$ and 9. A feasible schedule of the two other jobs must utilize these idle-segments.

LEMMA 3.1. *A k'-partition of $\mathcal{A}$ exists iff there is a feasible schedule of all jobs, using at most $k = k'-1$ preemptions per job.*

Proof. Assume there exists a k'-partition of $\mathcal{A}$, then we construct the following feasible schedule. We first assign all the tight jobs in their corresponding windows. Since $d_j - r_j = Z$ for each tight job, these jobs are processed with no preemptions. Their assignment leaves idle-segments of lengths $a_1, a_2, \ldots, a_{k'm}$. The remaining jobs will be assigned to these idle-segments in the following way. Let $S_1, \cdots, S_m$ be a partition of $\mathcal{A}$ such that for all $1 \leq j \leq m$, $\sum_{a \in S_j} a = Z$. For any subset $S_j = \{a_{j_1}, a_{j_2}, \cdots, a_{j_{k'}}\}$, schedule the jth job in the corresponding k' idle-segments of lengths $a_{j_1}, a_{j_2}, \cdots, a_{j_{k'}}$. Since $a_{j_1} + a_{j_2} + \cdots + a_{j_{k'}} = Z$, we have a feasible k-bounded preemptive schedule.

For the other side of the reduction, suppose we have a feasible k-bounded preemptive schedule of all jobs. First, since $d_j - r_j = Z$ for each tight job, these jobs must be processed with no preemptions, leaving $k'm$ idle-segments of total length Zm. The bounded preemptions imply that each of the other jobs is assigned on at most k' idle-segments. Since there are $k'm$ idle-segments and m jobs, and the total length of the idle-segments equals the sum of the processing times of the jobs, each job must be assigned to exactly k' holes of total length Z. Such a schedule induces a k'-partition of $\mathcal{A}$. ∎

For $k = 1$ we show a reduction from *equal-Partition*. Given a set of $2m$ numbers $\mathcal{A} = \{a_1, ..., a_{2m}\}$ such that $\sum_{i=1}^{2m} a_i = 2S$, the goal is to divide $\mathcal{A}$ into two subsets, each of cardinality m and total sum S. The equal-Partition problem is known to be NP-complete [10].

Given an instance of equal-Partition, we construct the following input for real-time 1-bounded preemptive scheduling. We first choose the uniform job size Z, such that Z is larger than the sum of any two numbers in the set $\mathcal{A}$. There are $4m+2$ jobs, all have the same processing time, Z. There are $2m+2$ tight jobs, for each of which $d_j - r_j = Z$. The windows of the tight jobs are defined such that their assignment leaves $2m+2$ idle-segments. The first $2m$ idle-segments have lengths $a_1, \ldots, a_{2m}$, and each of the two additional idle-segments have length $mZ - S$. For the $2m$ additional non-tight jobs, $r_j = 0$ and $d_j = \sum_j p_j = (4m+2)Z$.

Similar to the proof for $k > 1$, a feasible schedule corresponds to an equal-partition: All idle-segments must be utilized by the non-tight jobs, and since each of them must be scheduled in at most two pieces, and Z is larger than any two of the first $2m$ idle-segments, every non-tight job must utilize exactly one of the first $2m$ idle-segments and complete its process in one of the last two idle-segments. The utilization of the two last idle-segments induces an equal-partition of $\mathcal{A}$. It is easy to see also that an equal-partition induces a feasible schedule. ∎

For non-uniform-length jobs, we show that the problem is strongly NP-hard already for $k = 1$.

THEOREM 3.2. *For any $\epsilon > 0$, the problem of real-time k-bounded preemptive scheduling on a single machine is strongly NP-hard, for all $1 \leq k \leq |\mathcal{J}|^{1-\epsilon}$.*

Proof. We show a reduction from 3-Partition. Given a set of $3m$ numbers $\mathcal{A} = \{a_1, ..., a_{3m}\}$ and a number Z, such that $\forall i, a_i \in (\frac{Z}{4}, \frac{Z}{2})$ and $\sum_{i=1}^{3m} a_i = Zm$, the goal is to divide $\mathcal{A}$ into m subsets $S_1, \cdots, S_m \subseteq \mathcal{A}$ such that for all $1 \leq j \leq m$ $\sum_{a \in S_j} a = Z$. The 3-Partition problem is known to be strongly NP-compete[10]. We assume that $\min_i a_i > 1$ (otherwise, we can set $a'_i = a_i + 1$ and $Z' = Z + 3$, and the assumptions holds).

We construct the following input for the scheduling problem. There are $n = 3m(k+1) + m$ jobs. $3m$ jobs correspond to the elements of $\mathcal{A}$, and $(3km + m)$ jobs are tight. All tight jobs j have length 1 and must be processed along the specific interval $[r_j, d_j)$, where $d_j - r_j = 1$. There is one tight job for each of the m triples, and k tight jobs for each element. The tight jobs are defined such that their assignment leaves $3mk + m$ idle-segments on the machine: m idle-segments of length Z, and $3mk$ idle-segments of length 1. Formally, for $1 \leq i \leq m$, job i has $p_i = 1$ and it must be processed in $[i(Z+1) - 1, i(Z+1))$. For $m+1 \leq i \leq 3mk+m$, job i has $p_i = 1$ and it must be processed in $[m(Z+1)+2(i-m)-1, m(Z+1)+2(i-m))$.

The $3m$ element-jobs are defined as follows. For $3mk + m + 1 \leq i \leq 3mk + 4m$, let $p_i = a_{i-3mk-m} + k$, $r_i = 0$ and $d_i = m(Z+1) + 6mk$. Note that element-jobs can be assigned along the whole schedule, and that

$d_i = \sum_{j=1}^{n} p_j$. This implies that the machine is not idle throughout a feasible schedule that assigns all the jobs.

LEMMA 3.2. *A 3-partition of $\mathcal{A}$ exists iff there is a feasible schedule of all jobs, using at most k preemptions per job.*

Proof. Assume that a 3-partition of $\mathcal{A}$ exists, then we construct the following feasible schedule. We first assign all the tight jobs in their corresponding windows. As explained above, the assignment of the tight jobs leaves $3mk+m$ idle-segments on the machine. m idle-segments of length Z and $3mk$ idle-segments of length 1. The element jobs will be assigned to these idle-segments in the following way: Let $S_1, \cdots, S_m$ be a partition of $\mathcal{A}$ such that for all $1 \leq j \leq m$, $\sum_{a \in S_j} a = Z$. For any subset $S_j = \{a_{j_1}, a_{j_2}, a_{j_3}\}$, recall that a_{j_i} induces a job of length $a_{j_i} + k$. Assign this job on a_{j_i} processing units in the j's Z-length idle-segment, and on k arbitrary unit-length idle-segments. The resulting schedule is a feasible k-bounded preemptive schedule of the whole instance.

For the other side of the reduction, suppose we have a feasible k-bounded preemptive schedule of all jobs. First, since $d_j - r_j = p_j = 1$ for each tight job, these jobs must be processed with no preemptions, leaving $3mk+m$ idle-segments as described above. The bounded preemptions imply that each of the element jobs is assigned on at most $k+1$ idle-segments. Since $p_i > k+1$ for each element job (recall we assume that $a_i > 1$), no element job can utilize more than k unit-length idle-segments. Since there are $3m$ such jobs and $3mk$ unit-length idle-segments, each of them utilizes exactly k unit-length idle-segments. We conclude that a job of length $a_i + k$ must be assigned to exactly k unit-length idle-segments and a single Z-length idle-segment. The assignment of jobs to the Z-length idle-segments induces a 3-partition. ∎

To show that the proof holds for any $1 \leq k \leq n^{1-\varepsilon}$, recall that in the reduction, given a 3-partition instance with $3m$ elements, we construct an instance with $n = 3m(k+1)+m = 3m(k+\frac{4}{3})$ jobs. Thus, given $n > 1$ and $\varepsilon > 0$, setting $k \leq n^{1-\varepsilon}$ would still keep m polynomial in n. Consequently, we take the number of elements in the 3-partition instance, $3m$, to be the maximum value satisfying: $3m \leq \frac{n}{n^{1-\varepsilon}+\frac{4}{3}}$ for some $m \geq 1$. ∎

4 Approximation Algorithms

4.1 Unweighed Problem Lawler [17] studied our problem assuming the number of preemptions is unbounded, that is, $k = \infty$. For the unweighed case, Lawler presented an optimal polynomial-time algorithm, and for the weighed case he gave a pseudo-polynomial-time algorithm. Let $O_{u,\infty}$ be the value of an unweighed optimal solution, and let S_u be the set of $O_{u,\infty}$ jobs in an unweighed optimal solution output by Lawler's algorithm.

Our algorithm, $\mathcal{A}_c$, for the unweighed case, runs Lawler's algorithm and 'repairs' S_u to obtain a k-bounded preemptive schedule. Formally, $\mathcal{A}_c$ omits the infeasible jobs from the schedule, i.e., all jobs that were preempted at least $k+1$ times.

THEOREM 4.1. *$\mathcal{A}_c$ yields a $\frac{k+1}{k}$-approximation for unweighed real-time k-bounded preemptive scheduling.*

Proof. Denote by ALG_u the number of jobs in the resulting schedule. The optimal solution obtained by Lawler's algorithm has the property that a job is preempted only when another job is released [17]. Therefore, the total number of preemptions is at most $O_{u,\infty} - 1$. Hence, the number of infeasible jobs is at most $\frac{O_{u,\infty}-1}{k+1}$ and $ALG_u \geq O_{u,\infty} - \frac{O_{u,\infty}-1}{k+1} = \frac{kO_{u,\infty}+1}{k+1}$. Denote by $O_{u,k}$ the value of an optimal solution for unweighed real-time k-bounded preemptive scheduling. Clearly, $O_{u,\infty} \geq O_{u,k}$. Hence, we have $ALG_u \geq \frac{kO_{u,\infty}+1}{k+1} \geq \frac{k}{k+1}O_{u,\infty} \geq \frac{k}{k+1}O_{u,k}$. ∎

4.2 Maximum Utilization In the maximum utilization problem, for each job, j, $w_j = p_j$. Thus, maximizing the weight of processed jobs is equivalent to maximizing the utilization of the machine. In this section, we present a 4-approximation algorithm for the *maximum utilization k-bounded preemptive scheduling* problem. The algorithm consists of a preprocessing phase in which the jobs are sorted by their processing-times (or weights), and a scheduling phase in which every job is considered once, some of the jobs are scheduled and some are rejected.

Algorithm 1 - Alg_{MU}

1: Sort the jobs in non-increasing order of processing times, that is, $p_1 \geq p_2 \geq \cdots \geq p_n$.
2: Consider the jobs in the sorted order
3: **for** the current job j **do**
4: If j can be added to the schedule, that is, if the total length of at most $k+1$ idle-segments in $[r_j, d_j)$ is at least p_j, schedule j in the leftmost feasible way. Otherwise, reject j.
5: **end for**

A schedule in the *leftmost feasible way* is obtained by considering the idle-segments in j's window from left to right. If the job can fit into the first $k+1$ idle-segments, then we assign it to the leftmost ones. Otherwise, we replace the smallest idle-segment in this

set by the next one ($k+2$ from left) and repeat, until we find an adequate set, or until the job is rejected − if it cannot fit into the $k+1$ longest idle-segments.

We use in the analysis the following notation. Given a (partial) schedule, let $s = [t_1, t_2)$ be a continuous time-segment. During s, the machine alternates between being idle and busy. For $\alpha \leq 1$, we say that s is α-loaded if the total length of its busy-segments is $\alpha \cdot |s|$. If the machine is busy at time t_1 or t_2, then let $expand(s)$ denote the segment in which we extend s to the left and to right to include the whole busy-segments that include t_1 and t_2. If the machine is idle at t_1 and t_2 then $expand(s) = s$.

For an instance $\mathcal{J}$, let J_{in} and J_{out} denote the sets of jobs that Alg_{MU} schedule and reject, respectively. We first show that at the time job j is rejected, every busy-segment in the schedule is longer than p_j. Formally,

LEMMA 4.1. *Let $j \in J_{out}$, and consider the schedule at the time j is rejected. For every busy-segment b, it holds that $|b| \geq p_j$.*

Proof. Consider a busy-segment b at the time j is rejected. Let j' be the longest job among the jobs processed by the machine along b. Since jobs are considered in non-increasing order of length, at the time j' is considered, the machine is idle along b. Since j' is assigned in the leftmost feasible way, without intended preemptions, it must be that $p_{j'} \leq |b|$. Since j is considered after j', we have $p_j \leq p_{j'}$, thus, $p_j \leq |b|$. ∎

Let U denote the union of the expanded segments corresponding to windows of rejected jobs. Formally, $U = \bigcup_{j \in J_{out}} expand([r_j, d_j))$. We show that U is at least 1/3-loaded. In fact, we show a stronger claim: we show that U is 1/3-loaded even if we exclude from U busy-segments corresponding to jobs that were assigned in some segment $[t_1, t_2) \subset U$ after some job $j \in J_{out}$ for which $[t_1, t_2) \subseteq expand[r_j, d_j)$ was rejected. That is, we first scan U and remove all the busy-segments that are included in the expanded window of some rejected job, j, and belong to a job shorter than j.

Next, scan U from left to right. Let i be the next idle-segment in U. Assume first that i is fully included in the window of some rejected job, and let $j \in J_{out}$ be the longest rejected job whose window includes i. Since j is rejected, i is shorter than p_j. The idle-segment i is surrounded by two busy-segments that, by Lemma 4.1 and the exclusion of busy-segments of shorter jobs, are at least as long as p_j, thus, longer than i.

If i is not fully included in a single window of a rejected job, then it can split into two adjacent idle-segments. Specifically, if $i = [t_1, t_2)$ is not fully included in a single window of a rejected job, then there are two rejected jobs, j_1, j_2 with windows $[r_1, d_1)$ and $[r_2, d_2)$ respectively, such that $t_1 < d_1 < t_2$, $r_2 \leq d_1$, and $d_2 > t_2$. Split i into $i_1 = [t_1, d_1)$ and $i_2 = [d_1, t_2)$. Since both j_1 and j_2 were rejected, we have $|i_1| < p_{j_1}$ and $t_2 - r_2 < p_{j_2}$, thus, $|i_2| < p_{j_2}$. By Lemma 4.1, the busy-segment that precedes i_1 has length at least $p_{j_1} > |i_1|$, and the busy-segment that follows i_2 has length at least $p_{j_2} > |i_2|$.

We conclude that U consists of busy-segments and a collection of idle-segments − each adjacent to at least one busy-segment in U of at least the same length. We can therefore define a mapping $f : S_I \to S_B$ from the collection of idle-segments in U to the busy-segments in U. The mapping f assigns every idle-segment to an adjacent busy-segment and by the above discussion, it fulfills the following properties: (i) For every $i \in S_I$, $|f(i)| \geq |i|$, and (ii) For every $b \in S_B$, $f(i) = b$ for either a single idle-segment adjacent to b, or for the two idle-segments surrounding b.

These properties imply the following:

COROLLARY 4.1. *U is at least 1/3-loaded.*

We can now conclude the analysis of Algorithm Alg_{MU}.

THEOREM 4.2. *For any $k > 0$, algorithm Alg_{MU} yields 4-approximation for the maximum utilization k-bounded preemptive scheduling problem.*

Proof. For an instance J, let J_{in} and J_{out} denote the sets of jobs that Alg_{MU} schedule and reject, respectively. Let $w(J_{in})$ be the total weight of the jobs in J_{in}, and recall that U is the union of the expanded windows corresponding to jobs in J_{out}. The utilization that an optimal algorithm can achieve from J_{out} is at most the union of J_{out}'s windows. Therefore, $OPT \leq w(J_{in}) + \cup_{j \in J_{out}} expand([r_j, d_j)) \leq w(J_{in}) + |U|$. By definition, $Alg_{MU} = w(J_{in})$. Using Corollary 4.1, we have that $|U| \leq 3w(J_{in})$. Hence, $OPT \leq 4w(J_{in}) = 4Alg_{MU}$. ∎

The following example shows that our analysis of Alg_{MU} is tight. The instance J consists of four jobs $\{a, b, c, d\}$, where $p_a = p_b = p_c = 1$ and $p_d = 1 + 2\varepsilon$. The windows of jobs a, b, c are $[0, 1), [1, 2), [2, 3)$, respectively, and the window of d is $[1 - \varepsilon, 4 + 2\varepsilon)$. For this instance, for any $k \geq 0$, Alg_{MU} assigns job d in time interval $[1 - \varepsilon, 2 + \varepsilon)$ and all other three jobs are rejected. An optimal solution assigns all four jobs to the machine (see Figure 1).

5 Experimental Results

We have implemented several heuristics for the problem and simulated their execution on instances of variable

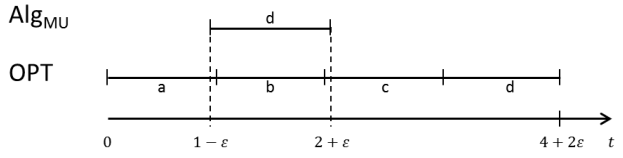

Figure 1: A tight example.

characteristics. The performance of the different heuristics was compared also to an upper bound on the optimal solution, calculated by solving a linear programming relaxation of our problem (given in Section 2). We describe below the heuristics and the experimental results. We distinguish between a single-machine environment, and an environment of identical parallel machines.

5.1 Experiments for a Single Machine In the single machine environment, we implemented three different heuristics, which consider the jobs in specific order. Therefore, each experiment is characterized by the heuristic used and the job order.

The jobs were sorted in four different ways: (i) non-decreasing order of processing-times ($p_1 \leq p_2 \leq \ldots \leq p_n$), (ii) non-increasing order of weights ($w_1 \geq w_2 \geq \ldots \geq w_n$), (iii) non-decreasing order of the ratio processing-time/weight ($\frac{p_1}{w_1} \leq \frac{p_2}{w_2} \leq \ldots \leq \frac{p_n}{w_n}$), and (iv) non-increasing order of loads ($l_1 \geq l_2 \geq \ldots \geq l_n$), where the load of job j is defined to be $l_j = \frac{p_j}{d_j - r_j}$.

The schedule S output in each experiment consists of a collection of pairs $\{j, P\}$, where j is a job and P is a set of at most $k+1$ pairs $\langle t, \ell \rangle$ such that $r_j \leq t \leq d_j - \ell$, $0 < \ell \leq p_j$, the intervals $[t, t+\ell)$ do not overlap, and $\sum_{\langle t,\ell \rangle \in P} \ell = p_j$; that is, S contains the set of intervals in which every job was processed.

The greedy heuristic simply adds jobs to the schedule, one after the other, in the leftmost feasible way (see Algorithm 2).

Note that the general greedy scheme coincides with our 4-approximation algorithm, Alg_{MU}, when run on an instance of the maximum utilization problem, and the jobs are considered in non-increasing order of weights.

The next two heuristics give higher priority to idle-intervals that are less likely to be required by other jobs. Specifically, prior to considering job j, the load generated by j on the interval $[r_j, d_j)$ is $l_j = \frac{p_j}{d_j - r_j}$. Consequently, every idle-interval of the machine is associated with a demand, given by the sum of loads on it. Our heuristics give higher priority to intervals having low demand. The two heuristics differ in the way they invoke preemptions. In the first heuristic, after allocating an idle-interval of lowest demand, we extend the processing of the job to adjacent idle-interval, so as to minimize preemptions and to keep the machine with few idle-busy toggles. In the second heuristic, we simply allocate the job to at most $k+1$ idle-intervals with lowest demands.

Algorithm 2 - General Greedy Scheme

1: $S = \emptyset$.
2: Use a certain rule to sort the jobs; consider the jobs in this order.
3: **for** the current job j **do**
4: If j can be added to the schedule, that is, if the total length of at most $k+1$ idle-segments in $[r_j, d_j)$ is at least p_j, then schedule j in the leftmost feasible way and update S. Otherwise, reject j.
5: **end for**
6: Return S.

Formally, let $s = [t_1, t_2)$ be a continuous time-segment on the machine. The *demand for s* is the sum of the loads of all jobs containing s in their window, i.e., $load(s) = \sum_{\{j | s \subseteq [r_j, d_j)\}} l_j$. The demand is calculated for a set of 'interesting' intervals, defined as follows. Initially, we scan the interval $[0, L)$ in which the machine is available and start a new interval at any time point which is a release time or a deadline of a job. Whenever a job is assigned, some intervals become unavailable, and the last interval (which may be partially allocated to the job) may be replaced by a shorter interval. The heuristics H_1 and H_2 differ only in Step 6: H_2 does not attempt to save on preemptions (see Algorithm 3).

In addition to the above heuristics, we have used the LP presented in Section 2, whose optimal solution yields an upper bound on the weight of an optimal schedule.

In each experiment we first determine the machine length L, and the maximal length and weight of a job, p_{max} and w_{max}, respectively. We then create the set of jobs $\mathcal{J}$ such that $[r_j, d_j) \subseteq [0, L)$ for all $j \in \mathcal{J}$. This is done by selecting at random for each job j, independently and uniformly, the values $p_j \in [1, p_{max}]$, $w_j \in [1, w_{max}]$, $r_j \in \{0, \ldots, L - p_j\}$ and $d_j \in \{r_j + p_j, \ldots, L\}$. Note that the heuristics as well as algorithm Alg_{MU} do not require integral parameters. In the experiments we assume that time is slotted, and for all jobs $j \in \mathcal{J}$, p_j, r_j and d_j are all integers. This enables a comparison with the value of the upper bound obtained by solving the linear program − in which these parameters are integers.

We tested the heuristics on a large set of instances, characterized also by their expected *density*. The density of an instance is given by $\frac{\sum_{j=1}^{n} p_j}{L}$. To create

instances with expected density D, we set the value of p_{max} and the number of jobs n such that $\frac{np_{max}}{2L} = D$.

In the first experiment, we examined the heuristics on instances of the maximum utilization problem, where for each job j, $p_j = w_j$. The three heuristics considered the jobs in non-increasing order of weights. Note that, with this order, the greedy heuristic is identical to Algorithm Alg_{MU}, analyzed in Section 4.2. We run all the algorithms as well as the LP on the same inputs on instances of different densities. In these runs, we set $L = 50$ and $p_{max} = 5$. The number of jobs was determined to achieve the desired density. The bound on number of preemptions was $k = 4$.

We run the experiment with each density value 25 times. The results (average of all runs) are presented in Figure 2. Note that in practice, the performance of greedy (Alg_{MU}) is much better than the theoretical 4-ratio. The heuristics H_1 and H_2 preform better than greedy for low densities, and are close to each other, while for very high density, greedy performs better.

Algorithm 3 - The heuristics H_1 and H_2

1: $S = \emptyset$
2: Use a certain rule to sort the jobs; consider the jobs in this order.
3: **for** the current job j **do**
4: If no interval in j's window is available, or if j was already scheduled in $k+1$ segments, reject j.
5: Let I be an interval with lowest demand in $j's$ window; schedule j in I.
6: (H_1) If j is not fully scheduled, and at least one of the two intervals adjacent to I is idle, schedule j on the one having minimal demand, and proceed in the same way. If both of the adjacent intervals are busy, go to step 4.
(H_2) If j is not fully scheduled, go to step 4.
7: Update S and the demands on all intervals.
8: **end for**
9: Return S.

In the second experiment we examined the heuristics on instances having uniform length jobs. As in the previous experiment, we run all heuristics and solved the LP on the same inputs, for instances with different densities. In all these runs, we set $L = 50$, $\forall j,\ p_j = 5$ and $w_{max} = 200$. The number of jobs was determined to achieve the desired density. The preemption bound was $k = 4$.

We run the experiment with each density value 25 times. The jobs were sorted either by weight, or by load. The results (average of all runs) are presented in Figure 3, which shows that our heuristics perform similarly. The main factor is the order by which the

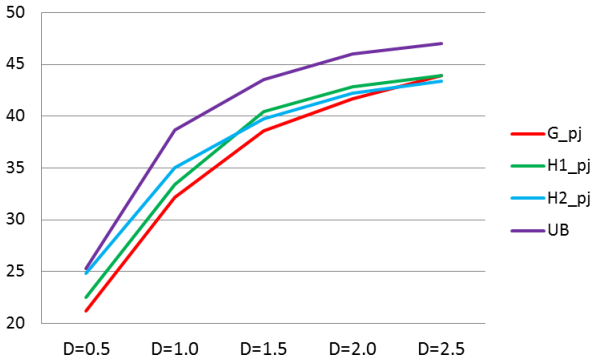

Figure 2: Maximum utilization instances

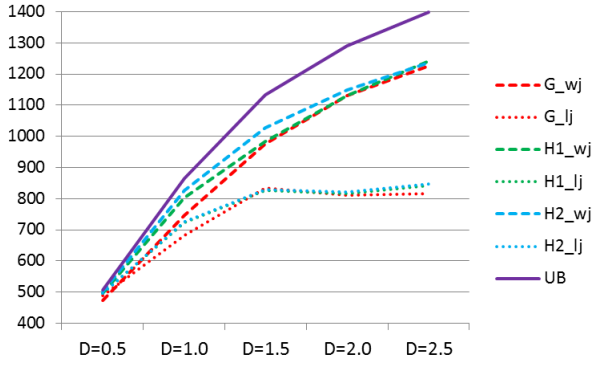

Figure 3: Instances with uniform processing times

jobs are considered. Sorting the jobs by weights is much better than prioritizing by the loads.

In the third experiment, we examined the heuristics on arbitrary instances. In these runs, we set $L = 1000$, $p_{max} = 16$ and $w_{max} = 200$. The number of jobs was correlated with the desired density, and the preemption bound was $k = 4$.

We run the experiment with each density value 100 times. The results (average of all runs) are presented in Figure 4. As in the previous experiment, the heuristics performance is mainly affected by the order in which the jobs are considered. Sorting the jobs in non-decreasing order of the ratio processing-time/weight yields the best performance. Comparing the different heuristics with the same order of jobs, we conclude that the best heuristic is H_2.

In the fourth experiment, we tested 'challenging' general instances, in which half of the jobs are tight (i.e., $p_j = d_j - r_j$) of length 1, and half have a time window of length at least twice their processing time ($2p_j \leq d_j - r_j$). We evaluated the schedule output by each algorithm on the same input, taking different values of k. The machine was available for $L = 1000$

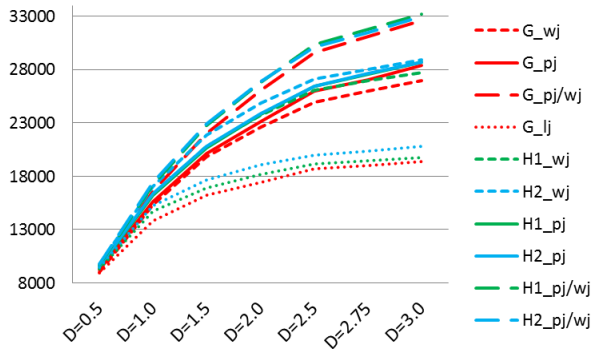

Figure 4: Arbitrary instances

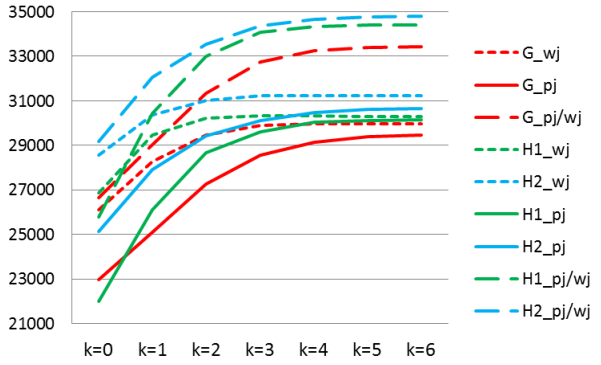

Figure 5: Challenging instances

time units. Also, $p_{max} = 33$, $w_{max} = 200$ and $n = 120$. Thus, the expected density was close to 2.

We run the experiment 100 times with each value of $0 \leq k \leq 6$. The results (average of all runs) are presented in Figure 5. In this experiment, again, sorting the jobs in non-decreasing p_j/w_j ratio yields best performance. The superiority of H_2 is more significant than in previous experiments.

For the next experiment we used a real data-set from workload logs collected from large scale parallel production systems (specifically, LPC-EGEE-2004-1.2-cln.swf in [22]). The results for $L = 2000, n = 1000$ and $k = 2$ are presented in Figure 7. The instance density is 1. The weights were chosen randomly between 1 and 10.

Note that here as well, the different heuristics perform similarly, and the main factor is the order by which the jobs are considered. In addition, H_1 and H_2 preform better than greedy, as in the simulated data.

5.2 Experiments for Parallel Machines In the environment of identical parallel machines, another main goal was to evaluate the power of migrations. We implemented three different heuristics. As in our experiments for a single machine, the heuristics consider the jobs in one of several possible orders.

The first heuristic, GP, allows no migrations. It applies the *General Greedy Scheme* in a 'first-fit' manner on the machines. Specifically, the considered job is assigned on the first machine that can accommodate it, using the general greedy scheme. A job is rejected if none of the machines can accommodate it.

The second heuristic, GPM, allows migrations. It examines the idle-segments on all machines and assigns the considered job on the leftmost available idle-segments, namely, job j is assigned at most $k+1$ disjoint idle-segments in $[r_j, d_j)$ whose total length is p_j.

The third heuristic, $GPMT$, allows job migration only above certain threshold. Specifically, job j is assigned to machine i only if, for some $\ell \geq 1$, machine i can process j for at least p_j/ℓ time units, using at most $\lfloor k/\ell \rfloor$ preemptions. We choose the minimum value of ℓ satisfying the condition. We proceed to the next machine with the remaining processing time and remaining budget of preemptions.

We have implemented the above heuristics and compared their performance on three different instances: (i) maximum utilization, where the jobs are considered in non-increasing order of processing-times, (ii) arbitrary instances, and (iii) challenging instances (see Section 5.1). In the two latter experiments, the jobs are considered in non-decreasing order of processing-time/weight ratio. In all of the experiments, we set the machine length L to 50, the number of machines to $m = 5$, the number of jobs to $n = 100$, and $w_{max} = 50$. We run the experiment with each value of $1 \leq k \leq 5$ 100 times. The results (average of all runs) are presented in Figure 6. In all the experiments, $GPMT$ outperforms the other two heuristics, or competitive with GPM on higher values of k. For low values of k, GP outperforms GPM, since the assignment of GPM leaves many short idle segments on the machines and thus requires more preemptions from jobs that are scheduled late. An important conclusion is that migrations are helpful, but not crucial for maximizing system utilization.

We tested our heuristics also on real data, (specifically, 'SDSC-BLUE-2000-4.2-cln.swf' in [22]). We used 8 machines, each available for $L = 2000$ time units. The input consisted of 8000 jobs (inducing density 8 on a single machine). As before, job time windows were selected randomly and uniformly in the available time span; job weights can take values in $[1, 5]$. In the first experiment, we examined the heuristics on a maximum utilization instance. In the second experiment, we examined the heuristics on the above instance, setting the preemption bound to be $k = 2$. The results are pre-

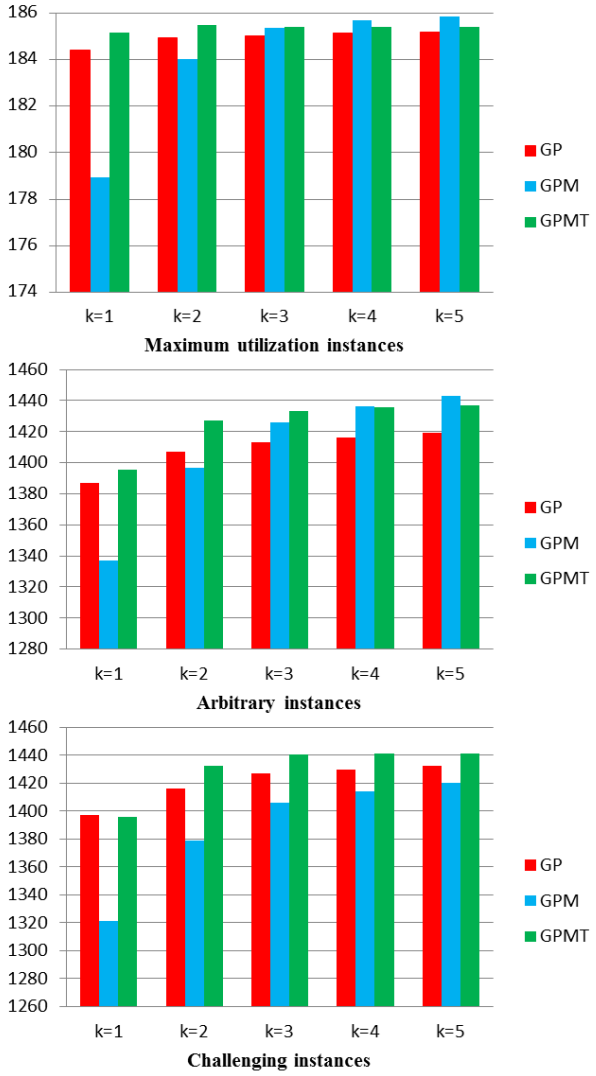

Figure 6: Results for parallel machines.

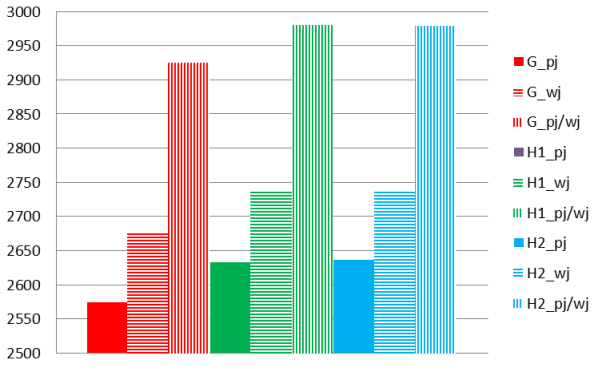

Figure 7: Real data single machine

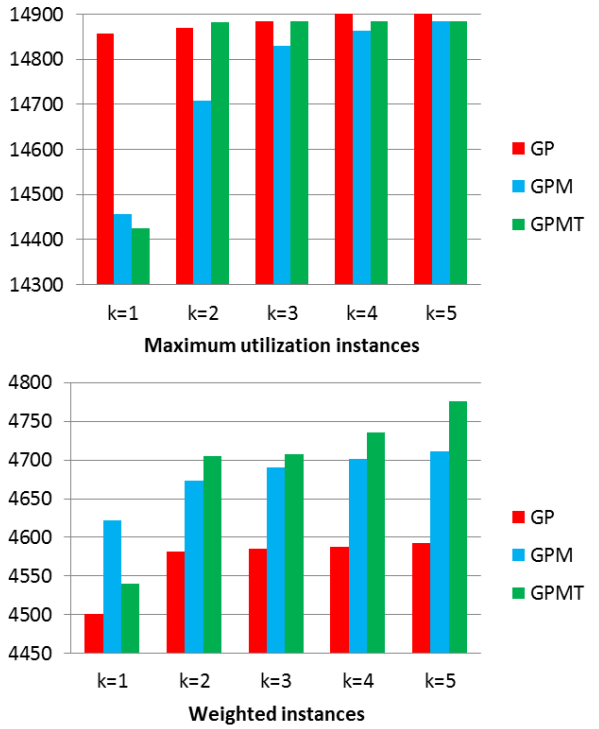

Figure 8: Real data parallel machines

sented in Figure 8. Note that also here, $GPMT$ outperforms the other two heuristics on the general instance, and achieves comparable performance on the maximum utilization instance.

References

[1] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2:245–252, 1999.

[2] P. Baptiste, M. Chrobak, C. Dürr, W. Jawor and N. Vakhania. Preemptive scheduling of equal-length jobs to maximize weighted throughput. *CoRR*, 2002.

[3] P. Baptiste, C. Le Pape and L. Péridy. Global Constraints for Partial CSPs: A case study of resource and due-date constraints. In Proc. of *Conf. on Principles and Practice of Constraint Programming*, 1998.

[4] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor and B. Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001.

[5] A. Bar-Noy, S. Guha, J. Naor and B. Schieber. Approximating the throughput of multiple machines in

real-time scheduling. *SIAM J. Comput.* 31(2):331–352, 2001.

[6] S.K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In Proc. of *ECRTS*, 137–144, 2005.

[7] P. Brucker and S. Knust. *Complexity results for scheduling Problems.* http://www.mathematik.uni-osnabrueck.de/research/OR/class, 2009.

[8] G. C. Buttazzo, M. Bertogna and G. Yao. Limited preemptive scheduling for real-time systems. A Survey. *IEEE Trans. Industrial Informatics*, 9(1):3–15, 2013.

[9] B. Chattopadhyay and S. K. Baruah. Limited-preemption scheduling on multiprocessors. In Proc. of *RTNS*, 2014.

[10] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[11] M. Grenier and N. Navet. Fine-tuning MAC-level protocols for optimized real-time QoS. *IEEE Trans. Industrial Informatics*, 4(1):6–15, 2008.

[12] J. Carlier. Problèmes d'ordonnancements à contraintes de ressources: *Algorithmes et Complexité.* Thèse d'Etat, 1984.

[13] S. Dauzère-Pérès. Minimizing late jobs in the general one-machine scheduling problem. *European Journal of Operational Research*, 81:134–142, 1995.

[14] T. Ebenlendr and J. Sgall. Optimal and online preemptive scheduling on uniformly related machines. In Proc. of *21st Symp. on Theoretical Aspects of Computer Science*, pp. 199–210, 2004.

[15] H. Kise, T. Ibaraki and H. Mine. A solvable case of the one-machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126, 1978.

[16] E.L. Lawler. Knapsack-like scheduling problems, the Moore-Hodgson algorithm and the 'Tower of Sets' property. *Mathl. Comput. Modelling*, 20(2):91–106, 1994.

[17] E.L. Lawler. A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. *Annals of Operations Research*, 26:125–133, 1990.

[18] M.J. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):102–109, 1968.

[19] L. Péridy, P. Baptiste and E. Pinson. Branch and bound method for the problem $1|r_i|\sum U_i$. In Proc. of *6th Int. Workshop on Project Management and Scheduling*, 1998.

[20] H. Shachnai, T. Tamir and G.J. Woeginger. Minimizing makespan and preemption costs on a system of uniform machines. *Algorithmica*, 42:309–334, 2005.

[21] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In Proc. of *Int. Conf. Real-Time Comput. Syst. Appl*, 1999.

[22] *The parallel workloads archive.* http://www.cs.huji.ac.il/labs/parallel/workload/logs.html, 2013.